This document describes use and implementation of a new fio histogram-based latency percentile measurement tool.

History

As a result of the need to measure I/O latency percentiles for a cluster with hundreds or even thousands of storage users, Karl Cronberg and I worked on the fiologparser_hist.py tool 2 years ago [1]. First he added the capability into fio to emit periodic latency histograms, which it already had been maintaining in memory, to a log file. Once he wrote the tool to postprocess this data, we then had a new ability to calculate latency percentiles as a function of time for a large distributed-storage cluster, so that we could understand how cluster events, such as node failures, impact response time of real applications. The results were pretty shocking - max latencies as high as hundreds of seconds during an OSD node failure. If you looked at fio's traditional latency percentile output, you might not see this because fio averages across entire time of test run and can't combine percentiles from multiple processes. This, plus customer-experience inputs to the Ceph team, helped to convince the Ceph team to introduce features that could help reduce these high latencies. We now are about to see if Ceph Luminous upstream release (RHCS 3) has better behavior in this area than Ceph Jewel (RHCS 2) upstream release did.

Why the old tool wasn't good enough

I got frustrated with [fiologparser_hist.py](), the tool for calculating fio latency percentiles as a function of time, because of:

- its dependency on python pandas module, which pulls in a LOT of packages
- its slowness (can take as long as an hour to process a set of histogram logs)
- hard to understand what it's doing
- hard to prove that it is generating the right results

I'm not aware of any other tool that does the same thing. So I rewrote it as fio-histo-log-pctiles.py. Differences between this implementation and the old one are:

- No dependency on numpy or pandas.
- about 50-100x faster
- It is ~263 LOC when you subtract comments, white space, and unit tests.
- unit tests exercise important subroutines within it and verify outputs.

AFAICT it is working well enough. When checked against fio's native percentile output, it seems to be giving very similar answers.   It is also giving similar answers to fiologparser_hist.py.

To see the syntax, just run “python fio-histo-log-pctiles.py --help”, it is fairly self-explanatory with the exception of the `--time-quantum` parameter, which is explained below.

Design and implementation

In summary, all input histogram logs are normalized to a fixed set of aligned time intervals (assumes good time synchronization across hosts), so that histograms can then be added directly to obtain cluster-wide histogram, and then latency percentiles are computed from that by summation.

Histogram log parsing

A fio histogram log consists of 3 columns of metadata followed by a fixed number of columns of histogram buckets. The 3 metadata columns are:

- **time_ms** - timestamp in milliseconds from start of test when this histogram's time interval began
- **direction** - 0 if the histogram is for reads, 1 if for writes.
- **bs** -- "block size", really I/O transfer size for test

`direction` : (read/write) is only useful at present for finding the end time of the histogram time interval (present in the next histogram record). The tool merges read and write histograms together at present. However, we need to keep this field because someday the tool might support separate read and write perf latency percentiles (yes they can be really different).

`time_ms` : Separate histogram records are emitted for reads and for writes on the same time interval. And they can be emitted in any order (read,write) or (write,read). Consequently, if we want to find the subsequent histogram record for that I/O direction (read or write) in order to get the end_time for the current record, we may have to read as many as 3 records farther to get it. For example, here's an excerpt from a real histogram log:

```
10203, 1, 4096, …
10203, 0, 4096, …
10601, 0, 4096, …
10601, 1, 4096, …
```

So the time interval for the write (1) and read (0) records is `[10203,10601]`, identical in this case but they don't have to be identical.

When we encounter the last histogram record (for an I/O direction) in the log, we can no longer get the end time from the next histogram record, since there isn't one, so we instead estimate the end time as the test end time in millisec.

`bs` - this field is really not used at all. There are 2 cases: either we are using a fixed I/O transfer size or a variable one. If the I/O transfer size is fixed, then it is in the command or job file used to run fio and is not needed in the result. If the I/O transfer size is variable, which fio supports, then the bs field is meaningless since a variety of I/O transfer sizes would have been used in a single histogram interval. So either way we can just ignore it.

Histogram buckets

The fio benchmark program already had histograms in memory -- these are described in stat.h line 38. The histogram is represented as an array of counters. The length of the array is the number of bucket groups times the number of buckets per group.

Histogram buckets are divided into groups. The size of a bucket group is given by `FIO_IO_U_PLAT_BITS` , the number of bits for identifying bucket index within the group. The default compile-time value is 6 bits, which means that we have 2^6 = 64 buckets per group. Each bucket within the group has the same time interval "width". Bucket groups 0 and 1 contain buckets with 1-nanosec time intervals (1-microsec for fio version 2), and each subsequent bucket groups double the time interval width. So for example bucket group 2 has a 2-nsec time width, bucket group 3 has a 4-nsec time interval, and so on.

The number of bucket groups is given by `FIO_IO_U_PLAT_GROUP_NR` , and the default is 29 for fio version 3 (19 for fio version 2). Hence the number of buckets in a histogram log record is 29*64 = 1856 (19*64 = 1216 for fio version 2).

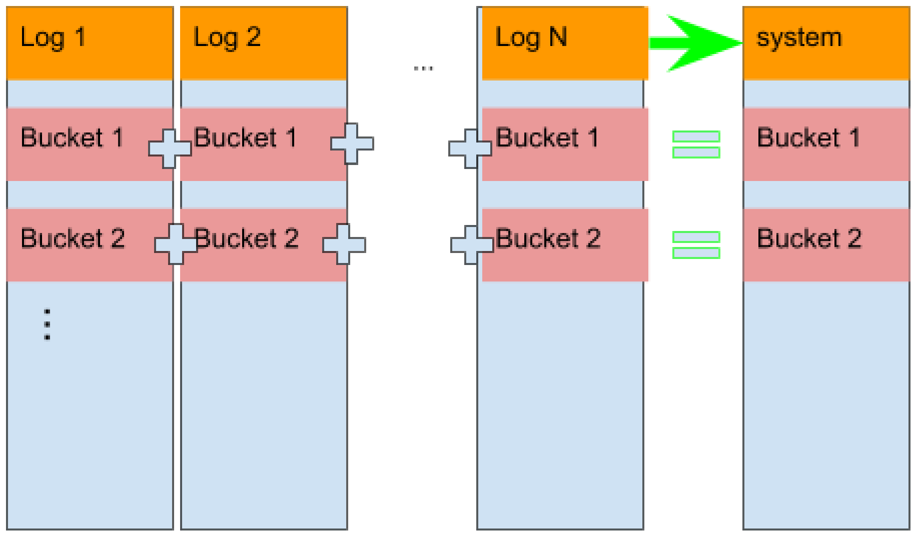
This is a lot of data so typically we do not generate histogram log records every second, but more like every 10 or even 60 seconds, depending on the length of the test. This is still much more frequent than normal fio behavior of generating one histogram for the entire test run (used to generate percentiles in fio output).

Bucket group	**Bucket 1 max latency (nsec)**	**Bucket 2 max latency (nsec)**	**...**	**Bucket 64 max latency (nsec)**
1	1	2		64
2	65	66		128
3	130	132		256
4	260	264		512
...				
29				17179869184

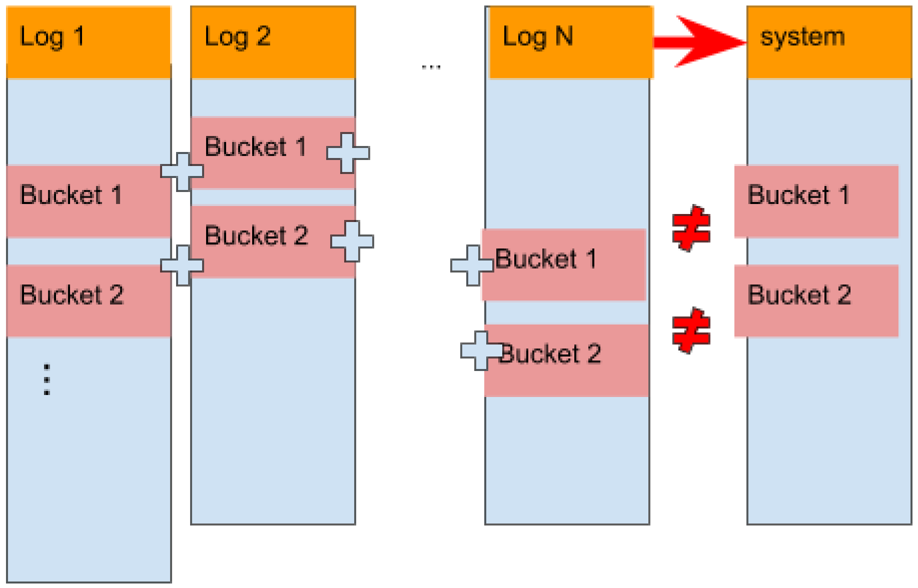
Note that the highest max latency for any bucket is 17.2 seconds = (64 * 2^28)/1000.0 . Any latency higher than that will be added to the bucket in the lower right corner. This is unfortunate for distributed systems where latencies can sometimes be much higher than 17 seconds due to network timeouts, etc.

Histogram time alignment

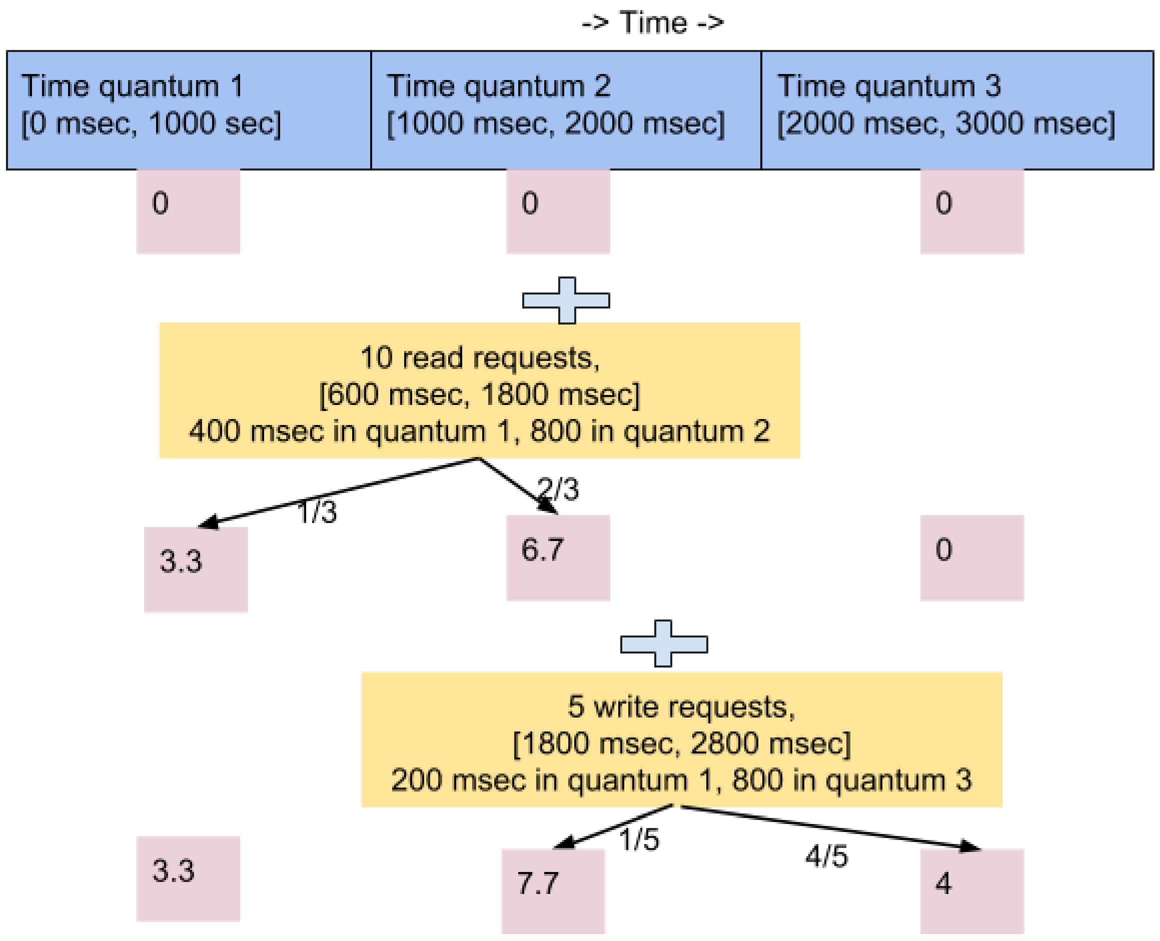
The key to this tool is the ability to add histograms from different fio processes together to obtain a system-wide or cluster-wide histogram that represents application latency. We should be able to do this because each histogram bucket is just a counter representing the number of I/O requests that fell within a latency range for that bucket.



However, unless these histograms are representing latencies from the *exact same* time interval, there is some degree of error in adding these raw histograms together.



This tool tries to minimize this error. based on the assumption that the `time_ms` field above is absolutely accurate (questionable in a distributed system or virtual machine environment). Even in a non-distributed fio run with multiple fio job processes, it is impossible for fio to precisely synchronize histogram log record writes down to the millisecond from all these jobs, although it tries to accomplish this. So this tool tries to compensate for this lack of synchronization. It does so by defining idealized "time quanta", fixed intervals of Q seconds starting at time 0.   It then takes the raw unsynchronized histograms described above, and fits them into these time quanta. If the raw histogram fits entirely within 1 time quantum, then it is just added to the histogram for that quantum. If the raw histogram overlaps multiple time quanta, then each bucket is weighted by the overlap with that time quantum and the raw histogram interval, using floating point arithmetic.



This time quantum alignment is done for each fio thread's histogram in the `align_histo_log` function. While this seems expensive to do, it drastically simplifies the merging of histograms from individual threads to aggregate histograms (could be node level or cluster level, whatever subset is of interest). The underlying assumption here is that I/O requests were done at a uniform rate within the time interval for that histogram log, so the weighted additions reflect what would have happened if we used a lower time resolution. We have no way to know that, so this is just an approximation.

Histogram addition

Once we have generated time-quantum-aligned histogram logs from the original raw logs, histogram k in each log is precisely time-aligned with histogram k in any other log, so it's ok to add them together to get an aggregate log, and this takes very little time. If python profiling shows that this is time consuming, we could use numpy to do vector addition of these arrays without much difficulty, avoiding python interpreter overhead.

Percentile calculation from a histogram

Now that we have a time-quantum-aligned histogram log for the entire cluster, the `get_pctiles` function computes percentiles in the usual way. Conceptually it is treating the histogram as if it was a probability distribution function, calculating the CDF (cumulative distribution function) by numerical integration, and then inverting this function to obtain the percentiles. This can be done to a good approximation by simple summation of the aggregate histogram, provided that we have enough histogram buckets. This is complicated slightly by the fact that fio bucket groups have different time interval widths, but we calculate the time intervals that go with each bucket in the histogram one time as described above, in the `time_ranges` function, and then just look up the time interval for each bucket by index. The algorithm, in an over-simplified form:

```
CDF[k] = sum(buckets[0:k])
CDF[-1] = 0.0
percentile_values = {}
For p in percentiles_wanted:
       find smallest j such that CDF[j] >= p
       offset_frac = (p - CDF[j-1])/(CDF[j] - CDF[j-1])
       interpolation = time_ranges[k].min +
                     (offset_frac * (time_ranges[j].max - time_ranges[j].min))
       percentile_values[p] = interpolation
```

References

[1] initial research by Cronberg and England from 2016 available upon request.